

Powerful Object-Oriented Programming

4th Edition
Covers Python 2.6 and 3.x

Learning

Python



Free Sampler

O'REILLY®

Mark Lutz

O'Reilly Ebooks—Your bookshelf on your devices!



When you buy an ebook through oreilly.com, you get lifetime access to the book, and whenever possible we provide it to you in four, DRM-free file formats—PDF, .epub, Kindle-compatible .mobi, and Android .apk ebook—that you can use on the devices of your choice. Our ebook files are fully searchable and you can cut-and-paste and print them. We also alert you when we've updated the files with corrections and additions.

Learn more at <http://oreilly.com/ebooks/>

You can also purchase O'Reilly ebooks through [iTunes](#), the [Android Marketplace](#), and [Amazon.com](#).

Learning Python, Fourth Edition

by Mark Lutz

Copyright © 2009 Mark Lutz. All rights reserved.
Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://my.safaribooksonline.com>). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Editor: Julie Steele

Production Editor: Sumita Mukherji

Copyeditor: Rachel Head

Production Services: Newgen North America

Indexer: John Bickelhaupt

Cover Designer: Karen Montgomery

Interior Designer: David Futato

Illustrator: Robert Romano

Printing History:

March 1999:	First Edition.
December 2003:	Second Edition.
October 2007:	Third Edition.
September 2009:	Fourth Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *Learning Python*, the image of a wood rat, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-0-596-15806-4

[M]

1280724660

[8/10]

Table of Contents

Preface	xxxi
---------------	------

Part I. Getting Started

1. A Python Q&A Session	3
Why Do People Use Python?	3
Software Quality	4
Developer Productivity	5
Is Python a “Scripting Language”?	5
OK, but What’s the Downside?	7
Who Uses Python Today?	7
What Can I Do with Python?	9
Systems Programming	9
GUIs	9
Internet Scripting	10
Component Integration	10
Database Programming	11
Rapid Prototyping	11
Numeric and Scientific Programming	11
Gaming, Images, Serial Ports, XML, Robots, and More	12
How Is Python Supported?	12
What Are Python’s Technical Strengths?	13
It’s Object-Oriented	13
It’s Free	13
It’s Portable	14
It’s Powerful	15
It’s Mixable	16
It’s Easy to Use	16
It’s Easy to Learn	17
It’s Named After Monty Python	17
How Does Python Stack Up to Language X?	17

Chapter Summary	18
Test Your Knowledge: Quiz	19
Test Your Knowledge: Answers	19
2. How Python Runs Programs	23
Introducing the Python Interpreter	23
Program Execution	24
The Programmer's View	24
Python's View	26
Execution Model Variations	29
Python Implementation Alternatives	29
Execution Optimization Tools	30
Frozen Binaries	32
Other Execution Options	33
Future Possibilities?	33
Chapter Summary	34
Test Your Knowledge: Quiz	34
Test Your Knowledge: Answers	34
3. How You Run Programs	35
The Interactive Prompt	35
Running Code Interactively	37
Why the Interactive Prompt?	38
Using the Interactive Prompt	39
System Command Lines and Files	41
A First Script	42
Running Files with Command Lines	43
Using Command Lines and Files	44
Unix Executable Scripts (#!)	46
Clicking File Icons	47
Clicking Icons on Windows	47
The input Trick	49
Other Icon-Click Limitations	50
Module Imports and Reloads	51
The Grander Module Story: Attributes	53
import and reload Usage Notes	56
Using exec to Run Module Files	57
The IDLE User Interface	58
IDLE Basics	58
Using IDLE	60
Advanced IDLE Tools	62
Other IDEs	63
Other Launch Options	64

Embedding Calls	64
Frozen Binary Executables	65
Text Editor Launch Options	65
Still Other Launch Options	66
Future Possibilities?	66
Which Option Should I Use?	66
Chapter Summary	68
Test Your Knowledge: Quiz	68
Test Your Knowledge: Answers	69
Test Your Knowledge: Part I Exercises	70

Part II. Types and Operations

4. Introducing Python Object Types	75
Why Use Built-in Types?	76
Python's Core Data Types	77
Numbers	78
Strings	80
Sequence Operations	80
Immutability	82
Type-Specific Methods	82
Getting Help	84
Other Ways to Code Strings	85
Pattern Matching	85
Lists	86
Sequence Operations	86
Type-Specific Operations	87
Bounds Checking	87
Nesting	88
Comprehensions	88
Dictionaries	90
Mapping Operations	90
Nesting Revisited	91
Sorting Keys: for Loops	93
Iteration and Optimization	94
Missing Keys: if Tests	95
Tuples	96
Why Tuples?	97
Files	97
Other File-Like Tools	99
Other Core Types	99
How to Break Your Code's Flexibility	100

User-Defined Classes	101
And Everything Else	102
Chapter Summary	103
Test Your Knowledge: Quiz	103
Test Your Knowledge: Answers	104
5. Numeric Types	105
Numeric Type Basics	105
Numeric Literals	106
Built-in Numeric Tools	108
Python Expression Operators	108
Numbers in Action	113
Variables and Basic Expressions	113
Numeric Display Formats	115
Comparisons: Normal and Chained	116
Division: Classic, Floor, and True	117
Integer Precision	121
Complex Numbers	122
Hexadecimal, Octal, and Binary Notation	122
Bitwise Operations	124
Other Built-in Numeric Tools	125
Other Numeric Types	127
Decimal Type	127
Fraction Type	129
Sets	133
Booleans	139
Numeric Extensions	140
Chapter Summary	141
Test Your Knowledge: Quiz	141
Test Your Knowledge: Answers	141
6. The Dynamic Typing Interlude	143
The Case of the Missing Declaration Statements	143
Variables, Objects, and References	144
Types Live with Objects, Not Variables	145
Objects Are Garbage-Collected	146
Shared References	148
Shared References and In-Place Changes	149
Shared References and Equality	151
Dynamic Typing Is Everywhere	152
Chapter Summary	153
Test Your Knowledge: Quiz	153
Test Your Knowledge: Answers	154

7. Strings	155
String Literals	157
Single- and Double-Quoted Strings Are the Same	158
Escape Sequences Represent Special Bytes	158
Raw Strings Suppress Escapes	161
Triple Quotes Code Multiline Block Strings	162
Strings in Action	163
Basic Operations	164
Indexing and Slicing	165
String Conversion Tools	169
Changing Strings	171
String Methods	172
String Method Examples: Changing Strings	174
String Method Examples: Parsing Text	176
Other Common String Methods in Action	177
The Original string Module (Gone in 3.0)	178
String Formatting Expressions	179
Advanced String Formatting Expressions	181
Dictionary-Based String Formatting Expressions	182
String Formatting Method Calls	183
The Basics	184
Adding Keys, Attributes, and Offsets	184
Adding Specific Formatting	185
Comparison to the % Formatting Expression	187
Why the New Format Method?	190
General Type Categories	193
Types Share Operation Sets by Categories	194
Mutable Types Can Be Changed In-Place	194
Chapter Summary	195
Test Your Knowledge: Quiz	195
Test Your Knowledge: Answers	196
8. Lists and Dictionaries	197
Lists	197
Lists in Action	200
Basic List Operations	200
List Iteration and Comprehensions	200
Indexing, Slicing, and Matrixes	201
Changing Lists In-Place	202
Dictionaries	207
Dictionaries in Action	209
Basic Dictionary Operations	209
Changing Dictionaries In-Place	210

More Dictionary Methods	211
A Languages Table	212
Dictionary Usage Notes	213
Other Ways to Make Dictionaries	216
Dictionary Changes in Python 3.0	217
Chapter Summary	223
Test Your Knowledge: Quiz	224
Test Your Knowledge: Answers	224
9. Tuples, Files, and Everything Else	225
Tuples	225
Tuples in Action	227
Why Lists and Tuples?	229
Files	229
Opening Files	230
Using Files	231
Files in Action	232
Other File Tools	238
Type Categories Revisited	239
Object Flexibility	241
References Versus Copies	241
Comparisons, Equality, and Truth	244
Python 3.0 Dictionary Comparisons	246
The Meaning of True and False in Python	246
Python's Type Hierarchies	248
Type Objects	249
Other Types in Python	250
Built-in Type Gotchas	251
Assignment Creates References, Not Copies	251
Repetition Adds One Level Deep	252
Beware of Cyclic Data Structures	252
Immutable Types Can't Be Changed In-Place	253
Chapter Summary	253
Test Your Knowledge: Quiz	254
Test Your Knowledge: Answers	254
Test Your Knowledge: Part II Exercises	255

Part III. Statements and Syntax

10. Introducing Python Statements	261
Python Program Structure Revisited	261
Python's Statements	262

A Tale of Two ifs	264
What Python Adds	264
What Python Removes	265
Why Indentation Syntax?	266
A Few Special Cases	269
A Quick Example: Interactive Loops	271
A Simple Interactive Loop	271
Doing Math on User Inputs	272
Handling Errors by Testing Inputs	273
Handling Errors with try Statements	274
Nesting Code Three Levels Deep	275
Chapter Summary	276
Test Your Knowledge: Quiz	276
Test Your Knowledge: Answers	277
11. Assignments, Expressions, and Prints	279
Assignment Statements	279
Assignment Statement Forms	280
Sequence Assignments	281
Extended Sequence Unpacking in Python 3.0	284
Multiple-Target Assignments	288
Augmented Assignments	289
Variable Name Rules	292
Expression Statements	295
Expression Statements and In-Place Changes	296
Print Operations	297
The Python 3.0 print Function	298
The Python 2.6 print Statement	300
Print Stream Redirection	302
Version-Neutral Printing	306
Chapter Summary	308
Test Your Knowledge: Quiz	308
Test Your Knowledge: Answers	308
12. if Tests and Syntax Rules	311
if Statements	311
General Format	311
Basic Examples	312
Multiway Branching	312
Python Syntax Rules	314
Block Delimiters: Indentation Rules	315
Statement Delimiters: Lines and Continuations	317
A Few Special Cases	318

Truth Tests	320
The if/else Ternary Expression	321
Chapter Summary	324
Test Your Knowledge: Quiz	324
Test Your Knowledge: Answers	324
13. while and for Loops	327
while Loops	327
General Format	328
Examples	328
break, continue, pass, and the Loop else	329
General Loop Format	329
pass	330
continue	331
break	331
Loop else	332
for Loops	334
General Format	334
Examples	335
Loop Coding Techniques	341
Counter Loops: while and range	342
Nonexhaustive Traversals: range and Slices	343
Changing Lists: range	344
Parallel Traversals: zip and map	345
Generating Both Offsets and Items: enumerate	348
Chapter Summary	349
Test Your Knowledge: Quiz	349
Test Your Knowledge: Answers	350
14. Iterations and Comprehensions, Part 1	351
Iterators: A First Look	351
The Iteration Protocol: File Iterators	352
Manual Iteration: iter and next	354
Other Built-in Type Iterators	356
List Comprehensions: A First Look	358
List Comprehension Basics	359
Using List Comprehensions on Files	359
Extended List Comprehension Syntax	361
Other Iteration Contexts	362
New Iterables in Python 3.0	366
The range Iterator	367
The map, zip, and filter Iterators	368
Multiple Versus Single Iterators	369

Dictionary View Iterators	370
Other Iterator Topics	372
Chapter Summary	372
Test Your Knowledge: Quiz	372
Test Your Knowledge: Answers	373
15. The Documentation Interlude	375
Python Documentation Sources	375
# Comments	376
The dir Function	376
Docstrings: __doc__	377
PyDoc: The help Function	380
PyDoc: HTML Reports	383
The Standard Manual Set	386
Web Resources	387
Published Books	387
Common Coding Gotchas	387
Chapter Summary	389
Test Your Knowledge: Quiz	389
Test Your Knowledge: Answers	390
Test Your Knowledge: Part III Exercises	390

Part IV. Functions

16. Function Basics	395
Why Use Functions?	396
Coding Functions	396
def Statements	398
def Executes at Runtime	399
A First Example: Definitions and Calls	400
Definition	400
Calls	400
Polymorphism in Python	401
A Second Example: Intersecting Sequences	402
Definition	402
Calls	403
Polymorphism Revisited	403
Local Variables	404
Chapter Summary	404
Test Your Knowledge: Quiz	405
Test Your Knowledge: Answers	405

17. Scopes	407
Python Scope Basics	407
Scope Rules	408
Name Resolution: The LEGB Rule	410
Scope Example	411
The Built-in Scope	412
The global Statement	414
Minimize Global Variables	415
Minimize Cross-File Changes	416
Other Ways to Access Globals	418
Scopes and Nested Functions	419
Nested Scope Details	419
Nested Scope Examples	419
The nonlocal Statement	425
nonlocal Basics	425
nonlocal in Action	426
Why nonlocal?	429
Chapter Summary	432
Test Your Knowledge: Quiz	433
Test Your Knowledge: Answers	434
18. Arguments	435
Argument-Passing Basics	435
Arguments and Shared References	436
Avoiding Mutable Argument Changes	438
Simulating Output Parameters	439
Special Argument-Matching Modes	440
The Basics	441
Matching Syntax	442
The Gritty Details	443
Keyword and Default Examples	444
Arbitrary Arguments Examples	446
Python 3.0 Keyword-Only Arguments	450
The min Wakeup Call!	453
Full Credit	454
Bonus Points	455
The Punch Line...	456
Generalized Set Functions	456
Emulating the Python 3.0 print Function	457
Using Keyword-Only Arguments	459
Chapter Summary	460
Test Your Knowledge: Quiz	461
Test Your Knowledge: Answers	462

19. Advanced Function Topics	463
Function Design Concepts	463
Recursive Functions	465
Summation with Recursion	465
Coding Alternatives	466
Loop Statements Versus Recursion	467
Handling Arbitrary Structures	468
Function Objects: Attributes and Annotations	469
Indirect Function Calls	469
Function Introspection	470
Function Attributes	471
Function Annotations in 3.0	472
Anonymous Functions: lambda	474
lambda Basics	474
Why Use lambda?	475
How (Not) to Obfuscate Your Python Code	477
Nested lambdas and Scopes	478
Mapping Functions over Sequences: map	479
Functional Programming Tools: filter and reduce	481
Chapter Summary	483
Test Your Knowledge: Quiz	483
Test Your Knowledge: Answers	483
20. Iterations and Comprehensions, Part 2	485
List Comprehensions Revisited: Functional Tools	485
List Comprehensions Versus map	486
Adding Tests and Nested Loops: filter	487
List Comprehensions and Matrixes	489
Comprehending List Comprehensions	490
Iterators Revisited: Generators	492
Generator Functions: yield Versus return	492
Generator Expressions: Iterators Meet Comprehensions	497
Generator Functions Versus Generator Expressions	498
Generators Are Single-Iterator Objects	499
Emulating zip and map with Iteration Tools	500
Value Generation in Built-in Types and Classes	506
3.0 Comprehension Syntax Summary	507
Comprehending Set and Dictionary Comprehensions	507
Extended Comprehension Syntax for Sets and Dictionaries	508
Timing Iteration Alternatives	509
Timing Module	509
Timing Script	510
Timing Results	511

Timing Module Alternatives	513
Other Suggestions	517
Function Gotchas	518
Local Names Are Detected Statically	518
Defaults and Mutable Objects	520
Functions Without returns	522
Enclosing Scope Loop Variables	522
Chapter Summary	522
Test Your Knowledge: Quiz	523
Test Your Knowledge: Answers	523
Test Your Knowledge: Part IV Exercises	524

Part V. Modules

21. Modules: The Big Picture	529
Why Use Modules?	529
Python Program Architecture	530
How to Structure a Program	531
Imports and Attributes	531
Standard Library Modules	533
How Imports Work	533
1. Find It	534
2. Compile It (Maybe)	534
3. Run It	535
The Module Search Path	535
Configuring the Search Path	537
Search Path Variations	538
The sys.path List	538
Module File Selection	539
Advanced Module Selection Concepts	540
Chapter Summary	541
Test Your Knowledge: Quiz	541
Test Your Knowledge: Answers	542
22. Module Coding Basics	543
Module Creation	543
Module Usage	544
The import Statement	544
The from Statement	545
The from * Statement	545
Imports Happen Only Once	546
import and from Are Assignments	546

Cross-File Name Changes	547
import and from Equivalence	548
Potential Pitfalls of the from Statement	548
Module Namespaces	550
Files Generate Namespaces	550
Attribute Name Qualification	552
Imports Versus Scopes	552
Namespace Nesting	553
Reloading Modules	554
reload Basics	555
reload Example	556
Chapter Summary	558
Test Your Knowledge: Quiz	558
Test Your Knowledge: Answers	558
23. Module Packages	561
Package Import Basics	561
Packages and Search Path Settings	562
Package <code>__init__.py</code> Files	563
Package Import Example	564
from Versus import with Packages	566
Why Use Package Imports?	566
A Tale of Three Systems	567
Package Relative Imports	569
Changes in Python 3.0	570
Relative Import Basics	570
Why Relative Imports?	572
The Scope of Relative Imports	574
Module Lookup Rules Summary	575
Relative Imports in Action	575
Chapter Summary	581
Test Your Knowledge: Quiz	582
Test Your Knowledge: Answers	582
24. Advanced Module Topics	583
Data Hiding in Modules	583
Minimizing from * Damage: <code>_X</code> and <code>__all__</code>	584
Enabling Future Language Features	584
Mixed Usage Modes: <code>__name__</code> and <code>__main__</code>	585
Unit Tests with <code>__name__</code>	586
Using Command-Line Arguments with <code>__name__</code>	587
Changing the Module Search Path	590
The as Extension for import and from	591

Modules Are Objects: Metaprograms	591
Importing Modules by Name String	594
Transitive Module Reloads	595
Module Design Concepts	598
Module Gotchas	599
Statement Order Matters in Top-Level Code	599
from Copies Names but Doesn't Link	600
from * Can Obscure the Meaning of Variables	601
reload May Not Impact from Imports	601
reload, from, and Interactive Testing	602
Recursive from Imports May Not Work	603
Chapter Summary	604
Test Your Knowledge: Quiz	604
Test Your Knowledge: Answers	605
Test Your Knowledge: Part V Exercises	605

Part VI. Classes and OOP

25. OOP: The Big Picture	611
Why Use Classes?	612
OOP from 30,000 Feet	613
Attribute Inheritance Search	613
Classes and Instances	615
Class Method Calls	616
Coding Class Trees	616
OOP Is About Code Reuse	619
Chapter Summary	622
Test Your Knowledge: Quiz	622
Test Your Knowledge: Answers	622
26. Class Coding Basics	625
Classes Generate Multiple Instance Objects	625
Class Objects Provide Default Behavior	626
Instance Objects Are Concrete Items	626
A First Example	627
Classes Are Customized by Inheritance	629
A Second Example	630
Classes Are Attributes in Modules	631
Classes Can Intercept Python Operators	633
A Third Example	634
Why Use Operator Overloading?	636
The World's Simplest Python Class	636

Classes Versus Dictionaries	639
Chapter Summary	641
Test Your Knowledge: Quiz	641
Test Your Knowledge: Answers	641
27. A More Realistic Example	643
Step 1: Making Instances	644
Coding Constructors	644
Testing As You Go	645
Using Code Two Ways	646
Step 2: Adding Behavior Methods	648
Coding Methods	649
Step 3: Operator Overloading	651
Providing Print Displays	652
Step 4: Customizing Behavior by Subclassing	653
Coding Subclasses	653
Augmenting Methods: The Bad Way	654
Augmenting Methods: The Good Way	654
Polymorphism in Action	656
Inherit, Customize, and Extend	657
OOP: The Big Idea	658
Step 5: Customizing Constructors, Too	658
OOP Is Simpler Than You May Think	660
Other Ways to Combine Classes	660
Step 6: Using Introspection Tools	663
Special Class Attributes	664
A Generic Display Tool	665
Instance Versus Class Attributes	666
Name Considerations in Tool Classes	667
Our Classes' Final Form	668
Step 7 (Final): Storing Objects in a Database	669
Pickles and Shelves	670
Storing Objects on a Shelf Database	671
Exploring Shelves Interactively	672
Updating Objects on a Shelf	674
Future Directions	675
Chapter Summary	677
Test Your Knowledge: Quiz	677
Test Your Knowledge: Answers	678
28. Class Coding Details	681
The class Statement	681
General Form	681

Example	682
Methods	684
Method Example	685
Calling Superclass Constructors	686
Other Method Call Possibilities	686
Inheritance	687
Attribute Tree Construction	687
Specializing Inherited Methods	687
Class Interface Techniques	689
Abstract Superclasses	690
Python 2.6 and 3.0 Abstract Superclasses	692
Namespaces: The Whole Story	693
Simple Names: Global Unless Assigned	693
Attribute Names: Object Namespaces	693
The “Zen” of Python Namespaces: Assignments Classify Names	694
Namespace Dictionaries	696
Namespace Links	699
Documentation Strings Revisited	701
Classes Versus Modules	703
Chapter Summary	703
Test Your Knowledge: Quiz	703
Test Your Knowledge: Answers	704
29. Operator Overloading	705
The Basics	705
Constructors and Expressions: <code>__init__</code> and <code>__sub__</code>	706
Common Operator Overloading Methods	706
Indexing and Slicing: <code>__getitem__</code> and <code>__setitem__</code>	708
Intercepting Slices	708
Index Iteration: <code>__getitem__</code>	710
Iterator Objects: <code>__iter__</code> and <code>__next__</code>	711
User-Defined Iterators	712
Multiple Iterators on One Object	714
Membership: <code>__contains__</code> , <code>__iter__</code> , and <code>__getitem__</code>	716
Attribute Reference: <code>__getattr__</code> and <code>__setattr__</code>	718
Other Attribute Management Tools	719
Emulating Privacy for Instance Attributes: Part 1	720
String Representation: <code>__repr__</code> and <code>__str__</code>	721
Right-Side and In-Place Addition: <code>__radd__</code> and <code>__iadd__</code>	723
In-Place Addition	725
Call Expressions: <code>__call__</code>	725
Function Interfaces and Callback-Based Code	727
Comparisons: <code>__lt__</code> , <code>__gt__</code> , and Others	728

The 2.6 <code>__cmp__</code> Method (Removed in 3.0)	729
Boolean Tests: <code>__bool__</code> and <code>__len__</code>	730
Object Destruction: <code>__del__</code>	732
Chapter Summary	733
Test Your Knowledge: Quiz	734
Test Your Knowledge: Answers	734
30. Designing with Classes	737
Python and OOP	737
Overloading by Call Signatures (or Not)	738
OOP and Inheritance: “Is-a” Relationships	739
OOP and Composition: “Has-a” Relationships	740
Stream Processors Revisited	742
OOP and Delegation: “Wrapper” Objects	745
Pseudoprivate Class Attributes	747
Name Mangling Overview	748
Why Use Pseudoprivate Attributes?	748
Methods Are Objects: Bound or Unbound	750
Unbound Methods are Functions in 3.0	752
Bound Methods and Other Callable Objects	754
Multiple Inheritance: “Mix-in” Classes	756
Coding Mix-in Display Classes	757
Classes Are Objects: Generic Object Factories	768
Why Factories?	769
Other Design-Related Topics	770
Chapter Summary	770
Test Your Knowledge: Quiz	770
Test Your Knowledge: Answers	771
31. Advanced Class Topics	773
Extending Built-in Types	773
Extending Types by Embedding	774
Extending Types by Subclassing	775
The “New-Style” Class Model	777
New-Style Class Changes	778
Type Model Changes	779
Diamond Inheritance Change	783
New-Style Class Extensions	788
Instance Slots	788
Class Properties	792
<code>__getattr__</code> and Descriptors	794
Metaclasses	794
Static and Class Methods	795

Why the Special Methods?	795
Static Methods in 2.6 and 3.0	796
Static Method Alternatives	798
Using Static and Class Methods	799
Counting Instances with Static Methods	800
Counting Instances with Class Methods	802
Decorators and Metaclasses: Part 1	804
Function Decorator Basics	804
A First Function Decorator Example	805
Class Decorators and Metaclasses	807
For More Details	808
Class Gotchas	808
Changing Class Attributes Can Have Side Effects	808
Changing Mutable Class Attributes Can Have Side Effects, Too	810
Multiple Inheritance: Order Matters	811
Methods, Classes, and Nested Scopes	812
Delegation-Based Classes in 3.0: <code>__getattr__</code> and built-ins	814
“Overwrapping-itis”	814
Chapter Summary	815
Test Your Knowledge: Quiz	815
Test Your Knowledge: Answers	815
Test Your Knowledge: Part VI Exercises	816

Part VII. Exceptions and Tools

32. Exception Basics	825
Why Use Exceptions?	825
Exception Roles	826
Exceptions: The Short Story	827
Default Exception Handler	827
Catching Exceptions	828
Raising Exceptions	829
User-Defined Exceptions	830
Termination Actions	830
Chapter Summary	833
Test Your Knowledge: Quiz	833
Test Your Knowledge: Answers	833
 33. Exception Coding Details	 835
The try/except/else Statement	835
try Statement Clauses	837
The try else Clause	839

Example: Default Behavior	840
Example: Catching Built-in Exceptions	841
The try/finally Statement	842
Example: Coding Termination Actions with try/finally	843
Unified try/except/finally	844
Unified try Statement Syntax	845
Combining finally and except by Nesting	845
Unified try Example	846
The raise Statement	848
Propagating Exceptions with raise	849
Python 3.0 Exception Chaining: raise from	849
The assert Statement	850
Example: Trapping Constraints (but Not Errors!)	851
with/as Context Managers	851
Basic Usage	852
The Context Management Protocol	853
Chapter Summary	855
Test Your Knowledge: Quiz	856
Test Your Knowledge: Answers	856
34. Exception Objects	857
Exceptions: Back to the Future	858
String Exceptions Are Right Out!	858
Class-Based Exceptions	859
Coding Exceptions Classes	859
Why Exception Hierarchies?	861
Built-in Exception Classes	864
Built-in Exception Categories	865
Default Printing and State	866
Custom Print Displays	867
Custom Data and Behavior	868
Providing Exception Details	868
Providing Exception Methods	869
Chapter Summary	870
Test Your Knowledge: Quiz	871
Test Your Knowledge: Answers	871
35. Designing with Exceptions	873
Nesting Exception Handlers	873
Example: Control-Flow Nesting	875
Example: Syntactic Nesting	875
Exception Idioms	877
Exceptions Aren't Always Errors	877

Functions Can Signal Conditions with raise	878
Closing Files and Server Connections	878
Debugging with Outer try Statements	879
Running In-Process Tests	880
More on sys.exc_info	881
Exception Design Tips and Gotchas	882
What Should Be Wrapped	882
Catching Too Much: Avoid Empty except and Exception	883
Catching Too Little: Use Class-Based Categories	885
Core Language Summary	885
The Python Toolset	886
Development Tools for Larger Projects	887
Chapter Summary	890
Test Your Knowledge: Quiz	891
Test Your Knowledge: Answers	891
Test Your Knowledge: Part VII Exercises	891

Part VIII. Advanced Topics

36. Unicode and Byte Strings	895
String Changes in 3.0	896
String Basics	897
Character Encoding Schemes	897
Python's String Types	899
Text and Binary Files	900
Python 3.0 Strings in Action	902
Literals and Basic Properties	902
Conversions	903
Coding Unicode Strings	904
Coding ASCII Text	905
Coding Non-ASCII Text	905
Encoding and Decoding Non-ASCII text	906
Other Unicode Coding Techniques	907
Converting Encodings	909
Coding Unicode Strings in Python 2.6	910
Source File Character Set Encoding Declarations	912
Using 3.0 Bytes Objects	913
Method Calls	913
Sequence Operations	914
Other Ways to Make bytes Objects	915
Mixing String Types	916
Using 3.0 (and 2.6) bytearray Objects	917

Using Text and Binary Files	920
Text File Basics	920
Text and Binary Modes in 3.0	921
Type and Content Mismatches	923
Using Unicode Files	924
Reading and Writing Unicode in 3.0	924
Handling the BOM in 3.0	926
Unicode Files in 2.6	928
Other String Tool Changes in 3.0	929
The re Pattern Matching Module	929
The struct Binary Data Module	930
The pickle Object Serialization Module	932
XML Parsing Tools	934
Chapter Summary	937
Test Your Knowledge: Quiz	937
Test Your Knowledge: Answers	937
37. Managed Attributes	941
Why Manage Attributes?	941
Inserting Code to Run on Attribute Access	942
Properties	943
The Basics	943
A First Example	944
Computed Attributes	945
Coding Properties with Decorators	946
Descriptors	947
The Basics	948
A First Example	950
Computed Attributes	952
Using State Information in Descriptors	953
How Properties and Descriptors Relate	955
__getattr__ and __getattribute__	956
The Basics	957
A First Example	959
Computed Attributes	961
__getattr__ and __getattribute__ Compared	962
Management Techniques Compared	963
Intercepting Built-in Operation Attributes	966
Delegation-Based Managers Revisited	970
Example: Attribute Validations	973
Using Properties to Validate	973
Using Descriptors to Validate	975
Using __getattr__ to Validate	977

Using <code>__getattr__</code> to Validate	978
Chapter Summary	979
Test Your Knowledge: Quiz	980
Test Your Knowledge: Answers	980
38. Decorators	983
What’s a Decorator?	983
Managing Calls and Instances	984
Managing Functions and Classes	984
Using and Defining Decorators	984
Why Decorators?	985
The Basics	986
Function Decorators	986
Class Decorators	990
Decorator Nesting	993
Decorator Arguments	994
Decorators Manage Functions and Classes, Too	995
Coding Function Decorators	996
Tracing Calls	996
State Information Retention Options	997
Class Blunders I: Decorating Class Methods	1001
Timing Calls	1006
Adding Decorator Arguments	1008
Coding Class Decorators	1011
Singleton Classes	1011
Tracing Object Interfaces	1013
Class Blunders II: Retaining Multiple Instances	1016
Decorators Versus Manager Functions	1018
Why Decorators? (Revisited)	1019
Managing Functions and Classes Directly	1021
Example: “Private” and “Public” Attributes	1023
Implementing Private Attributes	1023
Implementation Details I	1025
Generalizing for Public Declarations, Too	1026
Implementation Details II	1029
Open Issues	1030
Python Isn’t About Control	1034
Example: Validating Function Arguments	1034
The Goal	1034
A Basic Range-Testing Decorator for Positional Arguments	1035
Generalizing for Keywords and Defaults, Too	1037
Implementation Details	1040
Open Issues	1042

Decorator Arguments Versus Function Annotations	1043
Other Applications: Type Testing (If You Insist!)	1045
Chapter Summary	1046
Test Your Knowledge: Quiz	1047
Test Your Knowledge: Answers	1047
39. Metaclasses	1051
To Metaclass or Not to Metaclass	1052
Increasing Levels of Magic	1052
The Downside of “Helper” Functions	1054
Metaclasses Versus Class Decorators: Round 1	1056
The Metaclass Model	1058
Classes Are Instances of type	1058
Metaclasses Are Subclasses of Type	1061
Class Statement Protocol	1061
Declaring Metaclasses	1062
Coding Metaclasses	1063
A Basic Metaclass	1064
Customizing Construction and Initialization	1065
Other Metaclass Coding Techniques	1065
Instances Versus Inheritance	1068
Example: Adding Methods to Classes	1070
Manual Augmentation	1070
Metaclass-Based Augmentation	1071
Metaclasses Versus Class Decorators: Round 2	1073
Example: Applying Decorators to Methods	1076
Tracing with Decoration Manually	1076
Tracing with Metaclasses and Decorators	1077
Applying Any Decorator to Methods	1079
Metaclasses Versus Class Decorators: Round 3	1080
Chapter Summary	1084
Test Your Knowledge: Quiz	1084
Test Your Knowledge: Answers	1085

Part IX. Appendixes

A. Installation and Configuration	1089
B. Solutions to End-of-Part Exercises	1101
Index	1139

PART I

Getting Started

A Python Q&A Session

If you've bought this book, you may already know what Python is and why it's an important tool to learn. If you don't, you probably won't be sold on Python until you've learned the language by reading the rest of this book and have done a project or two. But before we jump into details, the first few pages of this book will briefly introduce some of the main reasons behind Python's popularity. To begin sculpting a definition of Python, this chapter takes the form of a question-and-answer session, which poses some of the most common questions asked by beginners.

Why Do People Use Python?

Because there are many programming languages available today, this is the usual question of newcomers. Given that there are roughly 1 million Python users out there at the moment, there really is no way to answer this question with complete accuracy; the choice of development tools is sometimes based on unique constraints or personal preference.

But after teaching Python to roughly 225 groups and over 3,000 students during the last 12 years, some common themes have emerged. The primary factors cited by Python users seem to be these:

Software quality

For many, Python's focus on readability, coherence, and software quality in general sets it apart from other tools in the scripting world. Python code is designed to be readable, and hence reusable and maintainable—much more so than traditional scripting languages. The uniformity of Python code makes it easy to understand, even if you did not write it. In addition, Python has deep support for more advanced software reuse mechanisms, such as object-oriented programming (OOP).

Developer productivity

Python boosts developer productivity many times beyond compiled or statically typed languages such as C, C++, and Java. Python code is typically one-third to one-fifth the size of equivalent C++ or Java code. That means there is less to type,

less to debug, and less to maintain after the fact. Python programs also run immediately, without the lengthy compile and link steps required by some other tools, further boosting programmer speed.

Program portability

Most Python programs run unchanged on all major computer platforms. Porting Python code between Linux and Windows, for example, is usually just a matter of copying a script's code between machines. Moreover, Python offers multiple options for coding portable graphical user interfaces, database access programs, web-based systems, and more. Even operating system interfaces, including program launches and directory processing, are as portable in Python as they can possibly be.

Support libraries

Python comes with a large collection of prebuilt and portable functionality, known as the *standard library*. This library supports an array of application-level programming tasks, from text pattern matching to network scripting. In addition, Python can be extended with both homegrown libraries and a vast collection of third-party application support software. Python's third-party domain offers tools for website construction, numeric programming, serial port access, game development, and much more. The NumPy extension, for instance, has been described as a free and more powerful equivalent to the Matlab numeric programming system.

Component integration

Python scripts can easily communicate with other parts of an application, using a variety of integration mechanisms. Such integrations allow Python to be used as a product customization and extension tool. Today, Python code can invoke C and C++ libraries, can be called from C and C++ programs, can integrate with Java and .NET components, can communicate over frameworks such as COM, can interface with devices over serial ports, and can interact over networks with interfaces like SOAP, XML-RPC, and CORBA. It is not a standalone tool.

Enjoyment

Because of Python's ease of use and built-in toolset, it can make the act of programming more pleasure than chore. Although this may be an intangible benefit, its effect on productivity is an important asset.

Of these factors, the first two (quality and productivity) are probably the most compelling benefits to most Python users.

Software Quality

By design, Python implements a deliberately simple and readable syntax and a highly coherent programming model. As a slogan at a recent Python conference attests, the net result is that Python seems to “fit your brain”—that is, features of the language interact in consistent and limited ways and follow naturally from a small set of core

concepts. This makes the language easier to learn, understand, and remember. In practice, Python programmers do not need to constantly refer to manuals when reading or writing code; it's a consistently designed system that many find yields surprisingly regular-looking code.

By philosophy, Python adopts a somewhat minimalist approach. This means that although there are usually multiple ways to accomplish a coding task, there is usually just one obvious way, a few less obvious alternatives, and a small set of coherent interactions everywhere in the language. Moreover, Python doesn't make arbitrary decisions for you; when interactions are ambiguous, explicit intervention is preferred over "magic." In the Python way of thinking, explicit is better than implicit, and simple is better than complex.*

Beyond such design themes, Python includes tools such as modules and OOP that naturally promote code reusability. And because Python is focused on quality, so too, naturally, are Python programmers.

Developer Productivity

During the great Internet boom of the mid-to-late 1990s, it was difficult to find enough programmers to implement software projects; developers were asked to implement systems as fast as the Internet evolved. Today, in an era of layoffs and economic recession, the picture has shifted. Programming staffs are often now asked to accomplish the same tasks with even fewer people.

In both of these scenarios, Python has shined as a tool that allows programmers to get more done with less effort. It is deliberately optimized for *speed of development*—its simple syntax, dynamic typing, lack of compile steps, and built-in toolset allow programmers to develop programs in a fraction of the time needed when using some other tools. The net effect is that Python typically boosts developer productivity many times beyond the levels supported by traditional languages. That's good news in both boom and bust times, and everywhere the software industry goes in between.

Is Python a “Scripting Language”?

Python is a general-purpose programming language that is often applied in scripting roles. It is commonly defined as an *object-oriented scripting language*—a definition that blends support for OOP with an overall orientation toward scripting roles. In fact, people often use the word “script” instead of “program” to describe a Python code file. In this book, the terms “script” and “program” are used interchangeably, with a slight

* For a more complete look at the Python philosophy, type the command `import this` at any Python interactive prompt (you'll see how in [Chapter 2](#)). This invokes an “Easter egg” hidden in Python—a collection of design principles underlying Python. The acronym EIBTI is now fashionable jargon for the “explicit is better than implicit” rule.

preference for “script” to describe a simpler top-level file and “program” to refer to a more sophisticated multifile application.

Because the term “scripting language” has so many different meanings to different observers, some would prefer that it not be applied to Python at all. In fact, people tend to make three very different associations, some of which are more useful than others, when they hear Python labeled as such:

Shell tools

Sometimes when people hear Python described as a scripting language, they think it means that Python is a tool for coding operating-system-oriented scripts. Such programs are often launched from console command lines and perform tasks such as processing text files and launching other programs.

Python programs can and do serve such roles, but this is just one of dozens of common Python application domains. It is not just a better shell-script language.

Control language

To others, scripting refers to a “glue” layer used to control and direct (i.e., script) other application components. Python programs are indeed often deployed in the context of larger applications. For instance, to test hardware devices, Python programs may call out to components that give low-level access to a device. Similarly, programs may run bits of Python code at strategic points to support end-user product customization without the need to ship and recompile the entire system’s source code.

Python’s simplicity makes it a naturally flexible control tool. Technically, though, this is also just a common Python role; many (perhaps most) Python programmers code standalone scripts without ever using or knowing about any integrated components. It is not just a control language.

Ease of use

Probably the best way to think of the term “scripting language” is that it refers to a simple language used for quickly coding tasks. This is especially true when the term is applied to Python, which allows much faster program development than compiled languages like C++. Its rapid development cycle fosters an exploratory, incremental mode of programming that has to be experienced to be appreciated.

Don’t be fooled, though—Python is not just for simple tasks. Rather, it makes tasks simple by its ease of use and flexibility. Python has a simple feature set, but it allows programs to scale up in sophistication as needed. Because of that, it is commonly used for quick tactical tasks and longer-term strategic development.

So, is Python a scripting language or not? It depends on whom you ask. In general, the term “scripting” is probably best used to describe the rapid and flexible mode of development that Python supports, rather than a particular application domain.

OK, but What's the Downside?

After using it for 17 years and teaching it for 12, the only downside to Python I've found is that, as currently implemented, its execution speed may not always be as fast as that of compiled languages such as C and C++.

We'll talk about implementation concepts in detail later in this book. In short, the standard implementations of Python today compile (i.e., translate) source code statements to an intermediate format known as *byte code* and then interpret the byte code. Byte code provides portability, as it is a platform-independent format. However, because Python is not compiled all the way down to binary machine code (e.g., instructions for an Intel chip), some programs will run more slowly in Python than in a fully compiled language like C.

Whether you will ever *care* about the execution speed difference depends on what kinds of programs you write. Python has been optimized numerous times, and Python code runs fast enough by itself in most application domains. Furthermore, whenever you do something “real” in a Python script, like processing a file or constructing a graphical user interface (GUI), your program will actually run at C speed, since such tasks are immediately dispatched to compiled C code inside the Python interpreter. More fundamentally, Python's speed-of-development gain is often far more important than any speed-of-execution loss, especially given modern computer speeds.

Even at today's CPU speeds, though, there still are some domains that do require optimal execution speeds. Numeric programming and animation, for example, often need at least their core number-crunching components to run at C speed (or better). If you work in such a domain, you can still use Python—simply split off the parts of the application that require optimal speed into *compiled extensions*, and link those into your system for use in Python scripts.

We won't talk about extensions much in this text, but this is really just an instance of the Python-as-control-language role we discussed earlier. A prime example of this dual language strategy is the *NumPy* numeric programming extension for Python; by combining compiled and optimized numeric extension libraries with the Python language, NumPy turns Python into a numeric programming tool that is efficient and easy to use. You may never need to code such extensions in your own Python work, but they provide a powerful optimization mechanism if you ever do.

Who Uses Python Today?

At this writing, the best estimate anyone can seem to make of the size of the Python user base is that there are roughly 1 million Python users around the world today (plus or minus a few). This estimate is based on various statistics, like download rates and developer surveys. Because Python is open source, a more exact count is difficult—there are no license registrations to tally. Moreover, Python is automatically included

with Linux distributions, Macintosh computers, and some products and hardware, further clouding the user-base picture.

In general, though, Python enjoys a large user base and a very active developer community. Because Python has been around for some 19 years and has been widely used, it is also very stable and robust. Besides being employed by individual users, Python is also being applied in real revenue-generating products by real companies. For instance:

- Google makes extensive use of Python in its web search systems, and employs Python's creator.
- The YouTube video sharing service is largely written in Python.
- The popular BitTorrent peer-to-peer file sharing system is a Python program.
- Google's popular App Engine web development framework uses Python as its application language.
- EVE Online, a Massively Multiplayer Online Game (MMOG), makes extensive use of Python.
- Maya, a powerful integrated 3D modeling and animation system, provides a Python scripting API.
- Intel, Cisco, Hewlett-Packard, Seagate, Qualcomm, and IBM use Python for hardware testing.
- Industrial Light & Magic, Pixar, and others use Python in the production of animated movies.
- JPMorgan Chase, UBS, Getco, and Citadel apply Python for financial market forecasting.
- NASA, Los Alamos, Fermilab, JPL, and others use Python for scientific programming tasks.
- iRobot uses Python to develop commercial robotic devices.
- ESRI uses Python as an end-user customization tool for its popular GIS mapping products.
- The NSA uses Python for cryptography and intelligence analysis.
- The IronPort email server product uses more than 1 million lines of Python code to do its job.
- The One Laptop Per Child (OLPC) project builds its user interface and activity model in Python.

And so on. Probably the only common thread amongst the companies using Python today is that Python is used all over the map, in terms of application domains. Its general-purpose nature makes it applicable to almost all fields, not just one. In fact, it's safe to say that virtually every substantial organization writing software is using Python, whether for short-term tactical tasks, such as testing and administration, or for long-term strategic product development. Python has proven to work well in both modes.

For more details on companies using Python today, see Python’s website at <http://www.python.org>.

What Can I Do with Python?

In addition to being a well-designed programming language, Python is useful for accomplishing real-world tasks—the sorts of things developers do day in and day out. It’s commonly used in a variety of domains, as a tool for scripting other components and implementing standalone programs. In fact, as a general-purpose language, Python’s roles are virtually unlimited: you can use it for everything from website development and gaming to robotics and spacecraft control.

However, the most common Python roles currently seem to fall into a few broad categories. The next few sections describe some of Python’s most common applications today, as well as tools used in each domain. We won’t be able to explore the tools mentioned here in any depth—if you are interested in any of these topics, see the Python website or other resources for more details.

Systems Programming

Python’s built-in interfaces to operating-system services make it ideal for writing portable, maintainable system-administration tools and utilities (sometimes called *shell tools*). Python programs can search files and directory trees, launch other programs, do parallel processing with processes and threads, and so on.

Python’s standard library comes with POSIX bindings and support for all the usual OS tools: environment variables, files, sockets, pipes, processes, multiple threads, regular expression pattern matching, command-line arguments, standard stream interfaces, shell-command launchers, filename expansion, and more. In addition, the bulk of Python’s system interfaces are designed to be portable; for example, a script that copies directory trees typically runs unchanged on all major Python platforms. The *Stackless Python* system, used by EVE Online, also offers advanced solutions to multiprocessing requirements.

GUIs

Python’s simplicity and rapid turnaround also make it a good match for graphical user interface programming. Python comes with a standard object-oriented interface to the Tk GUI API called *tkinter* (*Tkinter* in 2.6) that allows Python programs to implement portable GUIs with a native look and feel. Python/tkinter GUIs run unchanged on Microsoft Windows, X Windows (on Unix and Linux), and the Mac OS (both Classic and OS X). A free extension package, *PMW*, adds advanced widgets to the tkinter toolkit. In addition, the *wxPython* GUI API, based on a C++ library, offers an alternative toolkit for constructing portable GUIs in Python.

Higher-level toolkits such as *PythonCard* and *Dabo* are built on top of base APIs such as wxPython and tkinter. With the proper library, you can also use GUI support in other toolkits in Python, such as *Qt* with PyQt, *GTK* with PyGTK, *MFC* with PyWin32, *.NET* with IronPython, and *Swing* with Jython (the Java version of Python, described in [Chapter 2](#)) or JPype. For applications that run in web browsers or have simple interface requirements, both Jython and Python web frameworks and server-side CGI scripts, described in the next section, provide additional user interface options.

Internet Scripting

Python comes with standard Internet modules that allow Python programs to perform a wide variety of networking tasks, in client and server modes. Scripts can communicate over sockets; extract form information sent to server-side CGI scripts; transfer files by FTP; parse, generate, and analyze XML files; send, receive, compose, and parse email; fetch web pages by URLs; parse the HTML and XML of fetched web pages; communicate over XML-RPC, SOAP, and Telnet; and more. Python's libraries make these tasks remarkably simple.

In addition, a large collection of third-party tools are available on the Web for doing Internet programming in Python. For instance, the *HTMLGen* system generates HTML files from Python class-based descriptions, the *mod_python* package runs Python efficiently within the Apache web server and supports server-side templating with its Python Server Pages, and the Jython system provides for seamless Python/Java integration and supports coding of server-side applets that run on clients.

In addition, full-blown web development framework packages for Python, such as *Django*, *TurboGears*, *web2py*, *Pylons*, *Zope*, and *WebWare*, support quick construction of full-featured and production-quality websites with Python. Many of these include features such as object-relational mappers, a Model/View/Controller architecture, server-side scripting and templating, and AJAX support, to provide complete and enterprise-level web development solutions.

Component Integration

We discussed the component integration role earlier when describing Python as a control language. Python's ability to be extended by and embedded in C and C++ systems makes it useful as a flexible glue language for scripting the behavior of other systems and components. For instance, integrating a C library into Python enables Python to test and launch the library's components, and embedding Python in a product enables onsite customizations to be coded without having to recompile the entire product (or ship its source code at all).

Tools such as the *SWIG* and *SIP* code generators can automate much of the work needed to link compiled components into Python for use in scripts, and the *Cython* system allows coders to mix Python and C-like code. Larger frameworks, such as Python's COM support on Windows, the Jython Java-based implementation, the IronPython .NET-based implementation, and various CORBA toolkits for Python, provide alternative ways to script components. On Windows, for example, Python scripts can use frameworks to script Word and Excel.

Database Programming

For traditional database demands, there are Python interfaces to all commonly used relational database systems—Sybase, Oracle, Informix, ODBC, MySQL, PostgreSQL, SQLite, and more. The Python world has also defined a *portable database API* for accessing SQL database systems from Python scripts, which looks the same on a variety of underlying database systems. For instance, because the vendor interfaces implement the portable API, a script written to work with the free MySQL system will work largely unchanged on other systems (such as Oracle); all you have to do is replace the underlying vendor interface.

Python's standard `pickle` module provides a simple *object persistence* system—it allows programs to easily save and restore entire Python objects to files and file-like objects. On the Web, you'll also find a third-party open source system named *ZODB* that provides a complete object-oriented database system for Python scripts, and others (such as *SQLObject* and *SQLAlchemy*) that map relational tables onto Python's class model. Furthermore, as of Python 2.5, the in-process *SQLite* embedded SQL database engine is a standard part of Python itself.

Rapid Prototyping

To Python programs, components written in Python and C look the same. Because of this, it's possible to prototype systems in Python initially, and then move selected components to a compiled language such as C or C++ for delivery. Unlike some prototyping tools, Python doesn't require a complete rewrite once the prototype has solidified. Parts of the system that don't require the efficiency of a language such as C++ can remain coded in Python for ease of maintenance and use.

Numeric and Scientific Programming

The *NumPy* numeric programming extension for Python mentioned earlier includes such advanced tools as an array object, interfaces to standard mathematical libraries, and much more. By integrating Python with numeric routines coded in a compiled language for speed, NumPy turns Python into a sophisticated yet easy-to-use numeric programming tool that can often replace existing code written in traditional compiled languages such as FORTRAN or C++. Additional numeric tools for Python support

animation, 3D visualization, parallel processing, and so on. The popular *SciPy* and *ScientificPython* extensions, for example, provide additional libraries of scientific programming tools and use NumPy code.

Gaming, Images, Serial Ports, XML, Robots, and More

Python is commonly applied in more domains than can be mentioned here. For example, you can do:

- Game programming and multimedia in Python with the *pygame* system
- Serial port communication on Windows, Linux, and more with the *PySerial* extension
- Image processing with *PIL*, *PyOpenGL*, *Blender*, *Maya*, and others
- Robot control programming with the *PyRo* toolkit
- XML parsing with the *xml* library package, the *xmlrpcLib* module, and third-party extensions
- Artificial intelligence programming with neural network simulators and expert system shells
- Natural language analysis with the *NLTK* package

You can even play solitaire with the *PySol* program. You'll find support for many such fields at the PyPI websites, and via web searches (search Google or <http://www.python.org> for links).

Many of these specific domains are largely just instances of Python's component integration role in action again. Adding it as a frontend to libraries of components written in a compiled language such as C makes Python useful for scripting in a wide variety of domains. As a general-purpose language that supports integration, Python is widely applicable.

How Is Python Supported?

As a popular open source system, Python enjoys a large and active development community that responds to issues and develops enhancements with a speed that many commercial software developers would find remarkable (if not downright shocking). Python developers coordinate work online with a source-control system. Changes follow a formal *PEP* (Python Enhancement Proposal) protocol and must be accompanied by extensions to Python's extensive regression testing system. In fact, modifying Python today is roughly as involved as changing commercial software—a far cry from Python's early days, when an email to its creator would suffice, but a good thing given its current large user base.

The *PSF* (Python Software Foundation), a formal nonprofit group, organizes conferences and deals with intellectual property issues. Numerous Python conferences are held around the world; O'Reilly's OSCON and the PSF's PyCon are the largest. The former of these addresses multiple open source projects, and the latter is a Python-only event that has experienced strong growth in recent years. Attendance at PyCon 2008 nearly *doubled* from the prior year, growing from 586 attendees in 2007 to over 1,000 in 2008. This was on the heels of a 40% attendance increase in 2007, from 410 in 2006. PyCon 2009 had 943 attendees, a slight decrease from 2008, but a still very strong showing during a global recession.

What Are Python's Technical Strengths?

Naturally, this is a developer's question. If you don't already have a programming background, the language in the next few sections may be a bit baffling—don't worry, we'll explore all of these terms in more detail as we proceed through this book. For developers, though, here is a quick introduction to some of Python's top technical features.

It's Object-Oriented

Python is an object-oriented language, from the ground up. Its class model supports advanced notions such as polymorphism, operator overloading, and multiple inheritance; yet, in the context of Python's simple syntax and typing, OOP is remarkably easy to apply. In fact, if you don't understand these terms, you'll find they are much easier to learn with Python than with just about any other OOP language available.

Besides serving as a powerful code structuring and reuse device, Python's OOP nature makes it ideal as a scripting tool for object-oriented systems languages such as C++ and Java. For example, with the appropriate glue code, Python programs can subclass (specialize) classes implemented in C++, Java, and C#.

Of equal significance, OOP is an *option* in Python; you can go far without having to become an object guru all at once. Much like C++, Python supports both procedural and object-oriented programming modes. Its object-oriented tools can be applied if and when constraints allow. This is especially useful in tactical development modes, which preclude design phases.

It's Free

Python is completely free to use and distribute. As with other open source software, such as Tcl, Perl, Linux, and Apache, you can fetch the entire Python system's source code for free on the Internet. There are no restrictions on copying it, embedding it in your systems, or shipping it with your products. In fact, you can even sell Python's source code, if you are so inclined.

But don't get the wrong idea: "free" doesn't mean "unsupported." On the contrary, the Python online community responds to user queries with a speed that most commercial software help desks would do well to try to emulate. Moreover, because Python comes with complete source code, it empowers developers, leading to the creation of a large team of implementation experts. Although studying or changing a programming language's implementation isn't everyone's idea of fun, it's comforting to know that you can do so if you need to. You're not dependent on the whims of a commercial vendor; the ultimate documentation source is at your disposal.

As mentioned earlier, Python development is performed by a community that largely coordinates its efforts over the Internet. It consists of Python's creator—*Guido van Rossum*, the officially anointed Benevolent Dictator for Life (BDFL) of Python—plus a supporting cast of thousands. Language changes must follow a formal enhancement procedure and be scrutinized by both other developers and the BDFL. Happily, this tends to make Python more conservative with changes than some other languages.

It's Portable

The standard implementation of Python is written in portable ANSI C, and it compiles and runs on virtually every major platform currently in use. For example, Python programs run today on everything from PDAs to supercomputers. As a partial list, Python is available on:

- Linux and Unix systems
- Microsoft Windows and DOS (all modern flavors)
- Mac OS (both OS X and Classic)
- BeOS, OS/2, VMS, and QNX
- Real-time systems such as VxWorks
- Cray supercomputers and IBM mainframes
- PDAs running Palm OS, PocketPC, and Linux
- Cell phones running Symbian OS and Windows Mobile
- Gaming consoles and iPods
- And more

Like the language interpreter itself, the standard library modules that ship with Python are implemented to be as portable across platform boundaries as possible. Further, Python programs are automatically compiled to portable byte code, which runs the same on any platform with a compatible version of Python installed (more on this in the next chapter).

What that means is that Python programs using the core language and standard libraries run the same on Linux, Windows, and most other systems with a Python interpreter. Most Python ports also contain platform-specific extensions (e.g., COM support on Windows), but the core Python language and libraries work the same everywhere. As mentioned earlier, Python also includes an interface to the Tk GUI toolkit called tkinter (Tkinter in 2.6), which allows Python programs to implement full-featured graphical user interfaces that run on all major GUI platforms without program changes.

It's Powerful

From a features perspective, Python is something of a hybrid. Its toolset places it between traditional scripting languages (such as Tcl, Scheme, and Perl) and systems development languages (such as C, C++, and Java). Python provides all the simplicity and ease of use of a scripting language, along with more advanced software-engineering tools typically found in compiled languages. Unlike some scripting languages, this combination makes Python useful for large-scale development projects. As a preview, here are some of the main things you'll find in Python's toolbox:

Dynamic typing

Python keeps track of the kinds of objects your program uses when it runs; it doesn't require complicated type and size declarations in your code. In fact, as you'll see in [Chapter 6](#), there is no such thing as a type or variable declaration anywhere in Python. Because Python code does not constrain data types, it is also usually automatically applicable to a whole range of objects.

Automatic memory management

Python automatically allocates objects and reclaims ("garbage collects") them when they are no longer used, and most can grow and shrink on demand. As you'll learn, Python keeps track of low-level memory details so you don't have to.

Programming-in-the-large support

For building larger systems, Python includes tools such as modules, classes, and exceptions. These tools allow you to organize systems into components, use OOP to reuse and customize code, and handle events and errors gracefully.

Built-in object types

Python provides commonly used data structures such as lists, dictionaries, and strings as intrinsic parts of the language; as you'll see, they're both flexible and easy to use. For instance, built-in objects can grow and shrink on demand, can be arbitrarily nested to represent complex information, and more.

Built-in tools

To process all those object types, Python comes with powerful and standard operations, including concatenation (joining collections), slicing (extracting sections), sorting, mapping, and more.

Library utilities

For more specific tasks, Python also comes with a large collection of precoded library tools that support everything from regular expression matching to networking. Once you learn the language itself, Python’s library tools are where much of the application-level action occurs.

Third-party utilities

Because Python is open source, developers are encouraged to contribute precoded tools that support tasks beyond those supported by its built-ins; on the Web, you’ll find free support for COM, imaging, CORBA ORBs, XML, database access, and much more.

Despite the array of tools in Python, it retains a remarkably simple syntax and design. The result is a powerful programming tool with all the usability of a scripting language.

It’s Mixable

Python programs can easily be “glued” to components written in other languages in a variety of ways. For example, Python’s C API lets C programs call and be called by Python programs flexibly. That means you can add functionality to the Python system as needed, and use Python programs within other environments or systems.

Mixing Python with libraries coded in languages such as C or C++, for instance, makes it an easy-to-use frontend language and customization tool. As mentioned earlier, this also makes Python good at rapid prototyping; systems may be implemented in Python first, to leverage its speed of development, and later moved to C for delivery, one piece at a time, according to performance demands.

It’s Easy to Use

To run a Python program, you simply type it and run it. There are no intermediate compile and link steps, like there are for languages such as C or C++. Python executes programs immediately, which makes for an interactive programming experience and rapid turnaround after program changes—in many cases, you can witness the effect of a program change as fast as you can type it.

Of course, development cycle turnaround is only one aspect of Python’s ease of use. It also provides a deliberately simple syntax and powerful built-in tools. In fact, some have gone so far as to call Python “executable pseudocode.” Because it eliminates much of the complexity in other tools, Python programs are simpler, smaller, and more flexible than equivalent programs in languages like C, C++, and Java.

It's Easy to Learn

This brings us to a key point of this book: compared to other programming languages, the core Python language is remarkably easy to learn. In fact, you can expect to be coding significant Python programs in a matter of days (or perhaps in just hours, if you're already an experienced programmer). That's good news for professional developers seeking to learn the language to use on the job, as well as for end users of systems that expose a Python layer for customization or control.

Today, many systems rely on the fact that end users can quickly learn enough Python to tailor their Python customizations' code onsite, with little or no support. Although Python does have advanced programming tools, its core language will still seem simple to beginners and gurus alike.

It's Named After Monty Python

OK, this isn't quite a technical strength, but it does seem to be a surprisingly well-kept secret that I wish to expose up front. Despite all the reptile icons in the Python world, the truth is that Python creator Guido van Rossum named it after the BBC comedy series *Monty Python's Flying Circus*. He is a big fan of Monty Python, as are many software developers (indeed, there seems to almost be a symmetry between the two fields).

This legacy inevitably adds a humorous quality to Python code examples. For instance, the traditional "foo" and "bar" for generic variable names become "spam" and "eggs" in the Python world. The occasional "Brian," "ni," and "shrubbery" likewise owe their appearances to this namesake. It even impacts the Python community at large: talks at Python conferences are regularly billed as "The Spanish Inquisition."

All of this is, of course, very funny if you are familiar with the show, but less so otherwise. You don't need to be familiar with the series to make sense of examples that borrow references to Monty Python (including many you will see in this book), but at least you now know their root.

How Does Python Stack Up to Language X?

Finally, to place it in the context of what you may already know, people sometimes compare Python to languages such as Perl, Tcl, and Java. We talked about performance earlier, so here we'll focus on functionality. While other languages are also useful tools to know and use, many people find that Python:

- Is more powerful than Tcl. Python’s support for “programming in the large” makes it applicable to the development of larger systems.
- Has a cleaner syntax and simpler design than Perl, which makes it more readable and maintainable and helps reduce program bugs.
- Is simpler and easier to use than Java. Python is a scripting language, but Java inherits much of the complexity and syntax of systems languages such as C++.
- Is simpler and easier to use than C++, but it doesn’t often compete with C++; as a scripting language, Python typically serves different roles.
- Is both more powerful and more cross-platform than Visual Basic. Its open source nature also means it is not controlled by a single company.
- Is more readable and general-purpose than PHP. Python is sometimes used to construct websites, but it’s also widely used in nearly every other computer domain, from robotics to movie animation.
- Is more mature and has a more readable syntax than Ruby. Unlike Ruby and Java, OOP is an option in Python—Python does not impose OOP on users or projects to which it may not apply.
- Has the dynamic flavor of languages like SmallTalk and Lisp, but also has a simple, traditional syntax accessible to developers as well as end users of customizable systems.

Especially for programs that do more than scan text files, and that might have to be read in the future by others (or by you!), many people find that Python fits the bill better than any other scripting or programming language available today. Furthermore, unless your application requires peak performance, Python is often a viable alternative to systems development languages such as C, C++, and Java: Python code will be much less difficult to write, debug, and maintain.

Of course, your author has been a card-carrying Python evangelist since 1992, so take these comments as you may. They do, however, reflect the common experience of many developers who have taken time to explore what Python has to offer.

Chapter Summary

And that concludes the hype portion of this book. In this chapter, we’ve explored some of the reasons that people pick Python for their programming tasks. We’ve also seen how it is applied and looked at a representative sample of who is using it today. My goal is to teach Python, though, not to sell it. The best way to judge a language is to see it in action, so the rest of this book focuses entirely on the language details we’ve glossed over here.

The next two chapters begin our technical introduction to the language. In them, we’ll explore ways to run Python programs, peek at Python’s byte code execution model, and introduce the basics of module files for saving code. The goal will be to give you

just enough information to run the examples and exercises in the rest of the book. You won't really start programming per se until [Chapter 4](#), but make sure you have a handle on the startup details before moving on.

Test Your Knowledge: Quiz

In this edition of the book, we will be closing each chapter with a quick pop quiz about the material presented therein to help you review the key concepts. The answers for these quizzes appear immediately after the questions, and you are encouraged to read the answers once you've taken a crack at the questions yourself. In addition to these end-of-chapter quizzes, you'll find lab exercises at the end of each part of the book, designed to help you start coding Python on your own. For now, here's your first test. Good luck!

1. What are the six main reasons that people choose to use Python?
2. Name four notable companies or organizations using Python today.
3. Why might you *not* want to use Python in an application?
4. What can you do with Python?
5. What's the significance of the Python `import this` statement?
6. Why does "spam" show up in so many Python examples in books and on the Web?
7. What is your favorite color?

Test Your Knowledge: Answers

How did you do? Here are the answers I came up with, though there may be multiple solutions to some quiz questions. Again, even if you're sure you got a question right, I encourage you to look at these answers for additional context. See the chapter's text for more details if any of these responses don't make sense to you.

1. Software quality, developer productivity, program portability, support libraries, component integration, and simple enjoyment. Of these, the quality and productivity themes seem to be the main reasons that people choose to use Python.
2. Google, Industrial Light & Magic, EVE Online, Jet Propulsion Labs, Maya, ESRI, and many more. Almost every organization doing software development uses Python in some fashion, whether for long-term strategic product development or for short-term tactical tasks such as testing and system administration.
3. Python's downside is performance: it won't run as quickly as fully compiled languages like C and C++. On the other hand, it's quick enough for most applications, and typical Python code runs at close to C speed anyhow because it invokes

linked-in C code in the interpreter. If speed is critical, compiled extensions are available for number-crunching parts of an application.

4. You can use Python for nearly anything you can do with a computer, from website development and gaming to robotics and spacecraft control.
5. `import this` triggers an Easter egg inside Python that displays some of the design philosophies underlying the language. You'll learn how to run this statement in the next chapter.
6. "Spam" is a reference from a famous Monty Python skit in which people trying to order food in a cafeteria are drowned out by a chorus of Vikings singing about spam. Oh, and it's also a common variable name in Python scripts....
7. Blue. No, yellow!

Python Is Engineering, Not Art

When Python first emerged on the software scene in the early 1990s, it spawned what is now something of a classic conflict between its proponents and those of another popular scripting language, Perl. Personally, I think the debate is tired and unwarranted today—developers are smart enough to draw their own conclusions. Still, this is one of the most common topics I'm asked about on the training road, so it seems fitting to say a few words about it here.

The short story is this: *you can do everything in Python that you can in Perl, but you can read your code after you do it.* That's it—their domains largely overlap, but Python is more focused on producing readable code. For many, the enhanced readability of Python translates to better code reusability and maintainability, making Python a better choice for programs that will not be written once and thrown away. Perl code is easy to write, but difficult to read. Given that most software has a lifespan much longer than its initial creation, many see Python as a more effective tool.

The somewhat longer story reflects the backgrounds of the designers of the two languages and underscores some of the main reasons people choose to use Python. Python's creator is a mathematician by training; as such, he produced a language with a high degree of uniformity—its syntax and toolset are remarkably coherent. Moreover, like math, Python's design is orthogonal—most of the language follows from a small set of core concepts. For instance, once one grasps Python's flavor of polymorphism, the rest is largely just details.

By contrast, the creator of the Perl language is a linguist, and its design reflects this heritage. There are many ways to accomplish the same tasks in Perl, and language constructs interact in context-sensitive and sometimes quite subtle ways—much like natural language. As the well-known Perl motto states, "There's more than one way to do it." Given this design, both the Perl language and its user community have historically encouraged freedom of expression when writing code. One person's Perl code can be radically different from another's. In fact, writing unique, tricky code is often a source of pride among Perl users.

But as anyone who has done any substantial code maintenance should be able to attest, *freedom of expression is great for art, but lousy for engineering*. In engineering, we need a minimal feature set and predictability. In engineering, freedom of expression can lead to maintenance nightmares. As more than one Perl user has confided to me, the result of too much freedom is often code that is much easier to rewrite from scratch than to modify.

Consider this: when people create a painting or a sculpture, they do so for themselves for purely aesthetic purposes. The possibility of someone else having to change that painting or sculpture later does not enter into it. This is a critical difference between art and engineering. When people write software, they are not writing it for themselves. In fact, they are not even writing primarily for the computer. Rather, good programmers know that code is written for the next human being who has to read it in order to maintain or reuse it. If that person cannot understand the code, it's all but useless in a realistic development scenario.

This is where many people find that Python most clearly differentiates itself from scripting languages like Perl. Because Python's syntax model almost forces users to write readable code, Python programs lend themselves more directly to the full software development cycle. And because Python emphasizes ideas such as limited interactions, code uniformity and regularity, and feature consistency, it more directly fosters code that can be used long after it is first written.

In the long run, Python's focus on code quality in itself boosts programmer productivity, as well as programmer satisfaction. Python programmers can be creative, too, of course, and as we'll see, the language does offer multiple solutions for some tasks. At its core, though, Python encourages good engineering in ways that other scripting languages often do not.

At least, that's the common consensus among many people who have adopted Python. You should always judge such claims for yourself, of course, by learning what Python has to offer. To help you get started, let's move on to the next chapter.

Want to read more?

You can find this [book](#) at oreilly.com
in print or ebook format.

It's also available at your favorite book retailer,
including [iTunes](#), [the Android Market](#), [Amazon](#),
and [Barnes & Noble](#).



O'REILLY[®]

Spreading the knowledge of innovators

oreilly.com